Laboratorium 9 – MS SQL Server 2008
Temat: Środowisko CLR w MS SQL Server – UDA i UDA
Opracowanie: A.Dydejczyk

1. Realizacja CLR UDT w SQL Server
2. Realizacja CLR UDA w SQL Server

CLR UDT w MS SQL Server – C#

https://technet.microsoft.com/en-us/library/ms186366%28v=sql.105%29.aspx

https://msdn.microsoft.com/pl-pl/library/ms131120%28v=sql.105%29.aspx

https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration/database-objects/clr-integration-custom-attributes-for-clr-routines?view=sql-server-2017

CLR UDT jest zaimplementowany jako klasa lub struktura na platformie .NET Framework. W ramach SQL Server Microsoft wprowadził trzy nowe typy danych CLR UDT: **`hierarchyid, geography`** i **`geometry`**. Identyfikacja CLR UDT realizowanego przez klasę (strukturę), która implementuje UDT opisuje atrybut **`SqlUserDefinedType`**

```
SqlUserDefinedType [( własność udt [, ...])]
własność udt :: =
Format = { Native | UserDefined }
| MaxByteSize = n
| IsByteOrdered = { true | false }
| ValidationMethod = string
| IsFixedLength = { true | false }
| Nazwa = string
```

Atrybuty CLR UDT i ich własności

Atrybut	Właściwość	Wartość	Opis
Serializable	brak	brak	Wskazuje, że UDT może być serializowany i deserializowany.
SqlUserDefinedType	Format.Native	brak	Określa, że UDT korzysta z natywnego formatu serializacji. Format natywny jest najbardziej wydajnym formatem przy serializacji i deserializacji, ale wprowadza pewne ograniczenia. Można użyć tylko typów danych .NET przechowujących wartości (Char, Integer i tak dalej) jako pola. Nie można wykorzystać referencyjnych typów danych (String, Array i tak dalej).

SqlUserDefinedType	Format.UserDefined	brak	Określa, że UDT korzysta z formatu serializacjizdefiniowanego przez użytkownika. Gdy jest użyty, UDT musi implementować interfejs IBinarySerialize i należy przygotować metody Write() oraz Read() serializujące i deserializujące dane UDT.
SqlUserDefinedType	Format.UserDefined	brak	Określa, że UDT korzysta z formatu serializacji zdefiniowanego przez użytkownika. Gdy jest użyty, UDT musi implementować interfejs IBinarySerialize i należy przygotować metody Write() oraz Read() serializujące i deserializujące dane UDT.
SqlUserDefinedType	IsByteOrdered	true/false	Pozwala porównywać i sortować wartości UDT w oparciu o ich reprezentację binarną. Jest również wymagane, jeśli tworzymy indeksy na kolumnach zdefiniowanych jako typ CLR UDT.
SqlUserDefinedType	IsFixedLength	true/false	Powinna być ustawiona na true, jeśli serializowana instancja UDT ma stałą długość.
SqlUserDefinedType	MaxByteSize	<=8000 lub –1	Maksymalny rozmiar serializowanych instancji w bajtach. Wartość może być z zakresu od 1 do 8000 lub przyjąć –1, co oznacza maksymalny rozmiar 2,1 GB.

Do realizacji zadania wykorzystamy Visual Studio, typ danych zostanie utworzony w języku C# oraz zostanie umieszczony w bazie danych **testCLR**.

Po utworzeniu nowego projektu w Visual Studio, wybieramy bazę danych **testCLR** i tworzymy szablon realizujący funkcjonalność – **UDT**.
Poniżej odpowiedni kod, który należy umieścić w otwartym szablonie. Kod realizuje liczbę zespoloną i umożliwia dodanie dwóch liczb zespolonych poprzez wbudowaną metodę realizującą to działanie.

// Skrypt Lab09.01

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

// dyrektywy dla kompilatora]
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]


 // deklaracja struktury reprezentującej instancję UDT
public struct ComplexNumber : Inullable //  dziedziczy po interfejsie INullable, aby
//dopuszczalne było stosowanie metody IsNull
{
    private double _x; //real part
    private double _y; //imaginary part

 // Właściwości IsNull i Null są wymagane we wszystkich UDT
    public bool IsNull // określająca, czy instancja UDT ma wartość NULL
    {
        get { return m_Null; }
```

```
    }

    public static ComplexNumber Null // zwraca instancję NULL typu UDT
    {
        get
        {
            ComplexNumber h = new ComplexNumber();
            h.m_Null = true; return h;
        }
    }
// konstruktor pobierający dwie wartości typu Double, który tworzy z nich instancję UDT.
    public ComplexNumber(double x, double y)
    {
        _x = x;
        _y = y;
        m_Null = false;
    }

    public ComplexNumber(bool nothing)
    {
        this._x = this._y = 0;
        this.m_Null = true;
    }

    public double RealPart
    {
        get { return _x; }
        set { _x = value; }
    }

    public double ImaginaryPart
    {
        get { return _y; }
        set { _y = value; }
    }
 //metoda ToString (wymagana we wszystkich UDT), która  konwertuje wewnętrzne
//dane UDT na ciąg znaków

    public override string ToString()
    {
        return _x.ToString() + "+" + _y.ToString() + "i";
    }

 //metoda Parse (wymagana we wszystkich UDT), która przyjmuje wartość ciągu znaków
//z SQL Server i przetwarza je na liczbę zespoloną
    public static ComplexNumber Parse(SqlString s)
    {
        string value = s.Value;
        if (s.IsNull || value.Trim() == "")
            return Null;
        string xstr = value.Substring(0, value.IndexOf('+'));
        string ystr = value.Substring(value.IndexOf('+') + 1,
            value.Length - xstr.Length - 2);
        double xx = double.Parse(xstr);
        double yy = double.Parse(ystr);
        return new ComplexNumber(xx, yy);
    }

    // Dodawanie liczb zespolonych.
    public static ComplexNumber Add(ComplexNumber c1, ComplexNumber c2)
    {
```

```
        return new ComplexNumber(c1._x + c2._x, c1._y + c2._y);
    }
    // Private member
    private bool m_Null;
}
```

Poprawność utworzonego typu danych sprawdzimy w SSMS wykonując poniższy polecenia T-SQL

```
CREATE TABLE test ( complexField dbo.ComplexNumber);
INSERT INTO test (complexField) VALUES('25+52i');
SELECT complexField FROM test;
```

// konwersji danych binarnych do postaci napisu

```
SELECT complexField.ToString() FROM test;
```

Metody statyczne UDT (deklarowane ze słowem kluczowym **static** w C# (słowem kluczowym Shared w Visual Basic) są wywoływane z SQL Server : **ComplexNumber::[Add]**

```
INSERT INTO test values(ComplexNumber::[Add]('12+25i','25+12i'));
DROP TABLE test;
```

W drugim przykładzie tworzymy typ opisujący punkt w układzie współrzędnych. Wartości współrzędnych należą do zbioru liczb całkowitych dodatnich.

// Skrypt Lab09.02

```
using System;
using System.Data;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
[Serializable]
// natywnego sposobu formatowania oznacza, że będą używane tylko typy proste
[SqlUserDefinedType(Format.Native,
IsByteOrdered = true, // typ jest uporządkowany binarnie.
ValidationMethodName = "SprawdzPunkt")]// metoda sprawdzania poprawności wprowadzanych danych
public struct Punkt : INullable
{
      private bool is_Null; // zwraca prawdę w przypadku pozytywnej weryfikacji wartości NULL obiektu
      private Int32 _x;
      private Int32 _y;

      public bool IsNull
      {
            get
            { return (is_Null); }
      }

      public static Punkt Null
      {
```

```
            get
            {
                  Punkt pt = new Punkt();
                  pt.is_Null = true;
                  return pt;
            }
      }
// przeciążenie metody ToString, w której w przypadku wartości null obiektu zwracany jest napis NULL.
      public override string ToString()
      {
      if (this.IsNull)
            return "NULL";
      else
      {
            StringBuilder builder = new StringBuilder();
            builder.Append(_x);
            builder.Append(",");
            builder.Append(_y);
            return builder.ToString();
      }
}
// przekształca wprowadzane jako napis dane reprezentujące punkt na postać obiektu
// dyrektywa SqlMethod z atrybutem OnNullCall ustawianym na false, co powoduje, że nie będzie ona wywoływana
//dla wartości null obiektu
[SqlMethod(OnNullCall = false)]
public static Punkt Parse(SqlString s)
{
      if (s.IsNull)
            return Null;
      Punkt pt = new Punkt();
      string[] xy = s.Value.Split(",".ToCharArray());
      pt.X = Int32.Parse(xy[0]);
      pt.Y = Int32.Parse(xy[1]);
      if (!pt.SprawdzPunkt())
            throw new ArgumentException("Invalid XY coordinate values.");
      return pt;
}
 //ustawić wartości współrzędnych jako właściwości typu obiektowego z wykorzystaniem akcesorów get i set
public Int32 X
{
      get
            { return this._x; }
      set
      {
            Int32 temp = _x;
            _x = value;
            if (!SprawdzPunkt())
            {
                  _x = temp; throw new ArgumentException("Zła współrzędna X.");
            }
      }
}

public Int32 Y
{
      get
            { return this._y; }
      set
      {
```

```
            Int32 temp = _y;
            _y = value;
            if (!SprawdzPunkt())
            {
                  _y = temp;
                  throw new ArgumentException("Zła współrzędna X.");
            }
      }
}

private bool SprawdzPunkt() // metoda sprawdzania poprawności wprowadzanych danych
{
      if ((_x >= 0) && (_y >= 0))
      {
            return true;
      }
      else
      {
            return false;
      }
}
//wyznaczenie odległości pomiędzy punktem reprezentowanym przez strukturę a drugim, danym jako parametr
[SqlMethod(OnNullCall = false)]
      public Double OdlegloscOdXY(Int32 iX, Int32 iY)
      {
            return Math.Sqrt(Math.Pow(iX - _x, 2.0) + Math.Pow(iY - _y, 2.0));
      }
// odległość od początku układu współrzędnych
[SqlMethod(OnNullCall = false)]
      public Double Odleglosc()
      {
            return OdlegloscOdXY(0, 0);
      }
// odległość od innego punktu,  reprezentowanego przez parametr typu zgodnego ze zbudowaną strukturą
[SqlMethod(OnNullCall = false)]
      public Double OdlegloscOd(Punkt pFrom)
      {
            return OdlegloscOdXY(pFrom.X, pFrom.Y);
      }
}
```

Testujemy

```
CREATE TABLE dbo. Punkty
(ID int IDENTITY(1,1) PRIMARY KEY, PunktyXY Punkt)
GO
INSERT INTO Punkty VALUES (CONVERT(Punkt, '3,4'));
INSERT INTO Punkty VALUES (CONVERT(Punkt, '1,5'));
INSERT INTO Punkty VALUES (CAST ('1,99' AS Punkt));
GO
SELECT ID, PunktyXY FROM Punkty
```

	ID	PunktyXY
1	1	0x008000000380000004
2	2	0x008000000180000005
3	3	0x008000000180000063
4	4	0x008000000180000063

```
SELECT ID, PunktyXY.ToString() AS PunktyXY FROM Punkty;
GO
SELECT ID, CAST(PunktyXY AS varchar) FROM Punkty;
GO
SELECT ID, CONVERT(varchar, PunktyXY) FROM Punkty;
```

	ID	PunktyXY
1	1	3,4
2	2	1,5
3	3	1,99
4	4	1,99

```
DECLARE @PunktXY Punkt;
SET @PunktXY = (SELECT PunktyXY FROM Punkty WHERE ID = 2);
SELECT @PunktXY.ToString() AS PunktXY;
```

```
SELECT ID, PunktyXY.ToString() AS PunktyXY FROM Punkty WHERE
PunktyXY > CONVERT(Punkt, '2,2');
```

porównaniu poddawana jest wewnętrzna notacja binarna, co oznacza, że aby wyrażenie było prawdziwe, pierwsza współrzędna punktu musi być większa lub równa 2. Jeśli jest większa, druga współrzędna nie odgrywa roli; gdy jest równa, druga współrzędna musi być większa niż 2

```
SELECT ID, PunktyXY.ToString() AS PunktyXY FROM Punkty WHERE
PunktyXY.X < PunktyXY.Y;
```

CLR UDA w MS SQL Server

https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration-database-objects-user-defined-functions/clr-user-defined-aggregates?view=sql-server-2017

```
SqlUserDefinedAggregate [ ( aggregate-attribute [,...] ) ]
aggregate-attribute::=
Format = { Native | UserDefined }
IsInvariantToDuplicates = { true | false }
IsInvariantToNulls = { true | false }
IsInvariantToOrder = { true | false }
IsNullIfEmpty = { true | false }
| MaxByteSize = n
```

Funkcje agregujące definiowane przez użytkownika mają funkcjonalność podobną do wbudowanych funkcji agregujących (tj. SUM czy AVG), działają od razu na całym zbiorze danych, w odróżnieniu od przetwarzania element po elemencie. Funkcje agregujące CLR UDA mają dostęp do funkcjonalności .NET i mogą operować na typach danych numerycznych, znakowych, daty i czasu lub UDT. Wyróżniamy dwa rodzaje CLR UDA – **proste i zaawansowane**.

Proste CLR UDA (**Format.Native**) wymagają zdefiniowania czterech metod.

- **`public void Init( )`** - UDA wywołuje swoją metodę, gdy silnik SQL Server przygotowuje agregat. Kod tej metody może resetować poszczególne zmienne do stanu początkowego, inicjalizować bufory i wykonywać inne czynności inicjalizacyjne.

- **`public void Accumulate(input_type value)`** - wywoływana jest przy przetwarzaniu każdego wiersza, pozwalając na dołączanie przekazanych danych. Metoda **`Accumulate()`** może zwiększać licznik, dodawać wartość wiersza do sumy lub wykonywać inne bardzie złożone obliczenia na danych z wiersza.
- **`public void Merge(udagg_class value)`** - wywoływana jest, gdy SQL Server zdecyduje, by wykorzystać przetwarzanie równoległe do zakończenia tworzenia agregatu. Jeśli silnik zapytania MS Server zdecyduje, by zastosować przetwarzanie równoległe, tworzy wiele instancji UDA i wywołuje metodę **`Merge()`**, by połączyć wyniki w pojedynczy agregat.
- **`public return_type Terminate( )`** - końcowa metoda UDA. Jest wywoływana po przetworzeniu wszystkich wierszy i połączeniu wszystkich agregatów utworzonych przy przetwarzaniu równoległym. Metoda **`Terminate()`** zwraca końcowy wynik z agregatu do silnika zapytania

Zaawansowana CLR UDA (**Format.Native**)

- z atrybutem **StructLayout**, zawierającym wartość **LayoutKind.Sequential**, która wymusza sekwencyjną serializację struktury w przypadku UDA o wartości **Format = UserDefined**
- Atrybut **SqlUserDefinedAggregate** deklaruje właściwości.
  - **Format.UserDefined** wskazuje, że UDA implementuje metody serializacyjne za pomocą interfejsu **IbinarySerialize,** zawierający zarówno metodę **Read**, jak i **Write**. Trzeba przekazac do SQL Server informację, jak serializować dane przy korzystaniu z typów referencyjnych
  - **IsNullIfEmpty** jest ustawiona na **true**, co wskazuje, że jeśli do UDA nie zostaną przekazane żadne wiersze, zwrócona zostanie wartość NULL.
  - **MaxByteSize** jest ustawiona na –1, co pozwala na serializację UDA, jeśli jest on większy niż 8000 bajtów. (Ograniczenie serializacji do 8000 bajtów było ścisłym ograniczeniem w SQL Server 2005, co uniemożliwiało serializację dużych obiektów, takich jak **ArrayList** w UDA).
  - **IsInvariantToDuplicates -** Określa, czy agregat jest niezmienny dla duplikatów. Na przykład MAX i MIN są niezmienne dla duplikatów, a AVG i SUM nie.
  - **IsInvariantToNulls -** Określa, czy agregat jest niezmienny do wartości pustych. Na przykład MAX i MIN są niezmienne do wartości null, a COUNT nie (ponieważ wartości null są uwzględniane w liczniku).
  - **IsInvariantToOrder -** Określa, czy agregat jest niezmienny do kolejności wartości. Określenie true daje optymalizatorowi zapytań większą elastyczność w wyborze planu wykonania i może skutkować poprawą wydajności.

Funkcja agregująca zliczająca liczbę wartości znajdujących się w określonym przedziale. Do realizacji zadania wykorzystamy język C#, Visual Studio a nasz agregat umieścimy w bazie danych **testCLR**

// Skrypt Lab09.03

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
[Serializable]// konieczne jest zastosowanie dyrektyw Serializable, odpowiadającej za zapis i odczyt z
//wewnętrznej postaci binarnej
// stosowane są w obliczeniach typy natywne
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct uda_CountOfRange
{
    private SqlInt32 iNumRange;
    public void Init() // wykonywaną na początku każdej grupy rekordów
    { this.iNumRange = 0; }

// wykonywana dla każdego z rekordów grupy, a której parametrami są pola wykorzystywane do obliczeń
    public void Accumulate(SqlInt32 Value)
    {
        if ((Value) < 2 && (Value) > -2)
        { this.iNumRange += 1; }
    }

    // łączy wyniki obliczeń wykonywanych w procesach równoległych
   public void Merge(uda_CountOfRange Group)
    { this.iNumRange += Group.iNumRange; }

   // wykonywana na zakończenie każdego z bloków, a służąca do ostatecznego sformatowania wyniku
   public SqlInt32 Terminate()
    { return ((SqlInt32)this.iNumRange); }
       };
```

Testujemy:

```
CREATE TABLE table_aggr(    a1 int,   a2 [char] NOT NULL)
```

Wstawiamy wartości:

```
INSERT INTO table_aggr VALUES (1,'a')
```

i jeszcze kilka .... a potem testujemy

```
SELECT dbo.uda_CountOfRange(a1) FROM  table_aggr;
```

Kolejny agregat zlicza liczbę wartości negatywnych.

// Skrypt Lab09.04

```
using System;
using System.Collections.Generic;

using System.Data;

using System.Data.SqlTypes;

using System.Runtime.InteropServices;

using Microsoft.SqlServer.Server;
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct uda_CountOfNegatives
{
    private SqlInt32 iNumOfNegatives;
    public void Init()
    {
        this.iNumOfNegatives = 0;
    }
    public void Accumulate(SqlInt32 Value)
    {
        if ((Value) < 0)
        {
            this.iNumOfNegatives += 1;
        }
    }

    public void Merge(uda_CountOfNegatives Group)
    {
        this.iNumOfNegatives += Group.iNumOfNegatives;
    }

    public SqlInt32 Terminate()
    {
        return ((SqlInt32)this.iNumOfNegatives);
    }
};
```

Ostatni agregat obliczy medianę w zbiorze danych. Agregat zostanie zrealizowany w wersji **rozszerzonej** CLR UDA.

// Skrypt Lab09.05

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlTypes;
using System.Runtime.InteropServices;
using Microsoft.SqlServer.Server;
namespace Apress.Examples
{
    [Serializable]
    [Microsoft.SqlServer.Server.SqlUserDefinedAggregate(
    Format.UserDefined,
    IsNullIfEmpty = true,
    MaxByteSize = 8000)]
    [StructLayout(LayoutKind.Sequential)]
    public struct Median : IBinarySerialize
    {
        List<double> temp;
        public void Init()
        { this.temp = new List<double>(); }
        public void Accumulate(SqlDouble number)
        {
            if (!number.IsNull)
            { this.temp.Add(number.Value); }
        }
        public void Merge(Median group)
        { this.temp.InsertRange(this.temp.Count, group.temp); }
        public SqlDouble Terminate()
        {
            SqlDouble result = SqlDouble.Null;
            this.temp.Sort();//sortowanie
            int first, second;//indeksy dwóch srodkowych liczb
            if (this.temp.Count % 2 == 1)
            {//nieparzysta ilosc
                first = this.temp.Count / 2;
                second = first;
            }
            else
            {//parzysta ilosc
                first = this.temp.Count / 2;
                second = first + 1;
            }
            if (this.temp.Count > 0)//jesli są liczby policz mediane
            {
                result = (SqlDouble)(this.temp[first] + this.temp[second]) / 2.0;
            }
            return result;
        }
        #region IBinarySerialize Members
        // Własna metoda do wczytywania serializowanych danych.
        public void Read(System.IO.BinaryReader r)
        {   this.temp = new List<double>();
            int j = r.ReadInt32();
            for (int i = 0; i < j; i++)
            { this.temp.Add(r.ReadDouble()); }
        }
        // Własna metoda do zapisywania serializowanych danych.
        public void Write(System.IO.BinaryWriter w)

        {
            w.Write(this.temp.Count);
            foreach (double d in this.temp)
            { w.Write(d); }
        }
        #endregion
    }

}
```